The Smalltalk class hierarchy

As we have seen, classes in Smalltalk are arranged in the form of a tree.

The class above a given class in the hierarchy is its superclass; classes below are its subclasses.

As in other object-oriented languages class inherits attributes from its superclasses. These attributes include—

- instance variables
- methods

When each class has a single (immediate) superclass, the system is said to have simple inheritance.

(Some languages support multiple inheritance.)

The set of all messages that an object responds to is called its message protocol.

If a subclass defines a message with the same selector as a message of a superclass, the subclass is said to override the superclass’s method.

Messages to self and super: In Smalltalk, as in other languages, performing an operation may require > 1 method call.

Often, a method will send another message to the same object.

In this case, the message is sent to the pseudo-variable self.

For example, the isNegative method of a BankAccount class might send a message to itself to check whether its balance is negative:
A message sent to the pseudo-variable `super` is received by the superclass of `self`.

But a message sent to `self` can invoke a method of the current class or any of its superclasses.

So why would we ever need a message to `super`?

One common use of messages to `super` is initialization. For example, an initialization message to a complicated object needs to initialize several simpler subobjects.

The names of the initialization methods may conflict, and so it is necessary to refer to one by referring to `super`.

**Abstract superclasses**

When

- two classes share several attributes, but
- neither class should be a subclass of the other,

an *abstract superclass* can be created to encompass the common attributes.

Then both of the classes are made subclasses of the abstract superclass.

*Example:* [Liu, Ch. 5] Consider several implementations for *Tables*, which store items and allow them to be retrieved.

A *LinkTable* is implemented as a linked list.

An *ArrayTable* holds each item in an array element.

How might we search a *Table*?
search(item);
{
    start;
    loop while (not end and next • item)
    end loop;
    if end then return not_found
    else return found;
}

How many methods are called from this code?
In which class should each of them be implemented?

• _____ should be implemented in
• _____ should be implemented in
• _____ should be implemented in

Do we need two versions of the search method?

Then where should this method be placed?

• In ArrayTable and LinkTable?
• In SequentialTable?
• In Table?

OK, then, let’s say we also have a PerfectHashTable, where there are no collisions; each item maps directly to one position in the hash table.

Either the sought-after element is found there or it’s not found at all.
But _should_ there be a search method in class _Table_?

So we can use an _abstract method_—a method that is declared but has no body.

In Smalltalk, abstract methods are implemented by

```smalltalk
self subclassResponsibility
```

Should we also use other abstract methods, say for _start_, _next_, and/or _end_?

Notice that in Smalltalk, objects of abstract classes can be instantiated, but they cannot be used.

How does this compare with C++ or Java?

Read about _hook methods_ in Liu §5.6.

Some abstract classes implement concrete methods.

Class _magnitude_ provides a general protocol for comparing elements to see which comes before the other one.

It implements several messages for comparing—

- `< aMagnitude`
- `<= aMagnitude`
- `> aMagnitude`
- `>= aMagnitude`
- `between: min and: max`

However, "<" and "=" must be defined in each subclass. Doesn’t mean the same thing in all.
It also implements some messages for testing—

\[
\begin{align*}
\text{min: } & \text{aMagnitude} \\
\text{max: } & \text{aMagnitude}
\end{align*}
\]

These methods are implemented like—

\[
\text{max: } \text{aMagnitude} \\
\text{self } < \text{aMagnitude} \\
\text{ifTrue: } [^\text{aMagnitude}] \\
\text{ifFalse: } [^\text{self}]
\]

Magnitude is one of the linear measures classes in Smalltalk

\[
\begin{array}{c}
\text{Object} \\
\text{Magnitude} \\
\text{Date} \quad \text{Time} \quad \text{Number} \quad \text{Character} \\
\text{Integer} \quad \text{Fraction} \quad \text{Float} \\
\text{SmallInteger} \quad \text{LargeNegativeInteger} \\
\text{LargePositiveInteger}
\end{array}
\]

If we were to execute—

\[
3.14159 \ \text{max: } 2.71828
\]

which class’s max: method would be executed?

Whose < method would it call?

**Instance methods vs. class methods:**

- Instance methods—messages that instances understand.

  *Example:* (A point \((x, y)\) is denoted \(x@y\). )
+ delta
  "Answer a new point that is the sum of the receiver and delta (which is a Point or a Number.)"

  | deltaPoint |
deltaPoint := delta asPoint.
^x + deltaPoint x
@ (y + deltaPoint y)

• class methods—

Example:
fromString: aString

"Answer a new string that is a copy of the argument, aString."

  | newString |
1 to: aString size do:
  [:i | newString at: i put: (aString at: i)].
^newString.

Most class methods are used for instance creation or initialization of class variables.

Metaclasses

[Liu §3.11] In Smalltalk, everything is an object. This includes classes.

Thus, a class must be an instance of some class. Which class?

Suppose that all classes were instances of a single class called Class.

What would be the problem with this?

Suppose that each class was an instance of a different class.
Instead, Smalltalk defines *metaclasses*.

Each class is the sole instance of its metaclass.

If $y$ is a subclass of $x$, then $y$’s metaclass is a subclass of $x$’s metaclass:

Therefore, we have the following rules.

1. Every class is ultimately a subclass of class `Object`, except for `Object` itself, which has no superclass.

2. Every object is an instance of a class.

3. Every class is the instance of a metaclass.

   Metaclasses are called by the name of the corresponding class.

   Thus, the metaclass of *array* is called

   ```
   Array class
   ```

4. There is no third layer in the background constituting meta-metaclasses. There are only classes and metaclasses.

   Thus, “the buck stops” at metaclasses.

What do the following evaluate to?

1. `2 class`
2. `2 class class`
3. `2 class class class`

---

**The class `Object`**

An object consists of its *representation* (how it is laid out in memory) and its operations.

Any object can be inspected to find out what’s inside. For example,
Note the distinction between objects and *variables*.

- A variable is not an object.
- A variable cannot be inspected, or stored into an array.
- But the value bound to a variable can!

An object’s representation consists of zero or more fields for instance variables. The instance variables are partitioned into two groups:

- named instance variables, and
- indexed instance variables.

The named instance variables precede the indexed instance variables.

Some classes have no instance variables at all, e.g.,

Some classes have only named instance variables, e.g.,

Some classes have only indexed instance variables, e.g.,

Some classes have both named and indexed instance variables, e.g.,

**Bindings: assignments and parameter passing**

In a language like Pascal, C, or Ada, an assignment like \( a := b \)

- is interpreted as “copy \( b \) into \( a \),” and
- is implemented by copying the contents of \( b \) into the space occupied by \( a \).
This implies that \( a \) and \( b \) must be the same type, and, more importantly, the same size.

But Smalltalk employs \textit{dynamic binding}, not \textit{static binding}. This means that the type of object “stored” in a variable is determined at run time, not at compile time.

Therefore, \( a := b \)

- is interpreted as “bind \( a \) to the same object that \( b \) is bound to,” and
- is implemented by copying the reference stored in \( b \) into the (pointer-sized) memory cell \( a \).

Assignments, then, physically copy references, but \textit{do not copy the objects}.

\textit{Aliasing}

\cite{Liu §4.5} The fact that assignment copies a reference can sometimes lead to a situation where an assignment to one variable affects another variable.

Consider the \texttt{Rectangle} class. Rectangles can be created by specifying their \textit{origin} and \textit{corner}—

\[
\texttt{Rectangle origin: 40@80 extent: 10@20}.
\]

or by specifying their origin and \textit{corner}.

\textbf{Accessing methods for class} \texttt{Rectangle}
Messages like `contains:`, `containsPoint:`, and `intersects:` can be used to test relationships between rectangles and points.

Class `Point` defines instances that represent points on the display screen.

Points are usually created by the binary message “@” to a number, `15@75` but they may also be created using the instance-creation message of the `Point` class protocol:

```
Point x: 15 y: 75
```

Point’s protocol has messages for accessing points:

- `x` Answers the `x` coordinate
- `y` Answers the `y` coordinate
- `x:` Sets the `x` coordinate equal to `arg`
- `y:` Sets the `y` coordinate equal to `arg`

Consider this code:
r := Rectangle origin: 20@20 extent: 10@10.
p := r corner.
p x: 50 y: 50.
r
What is now the extent of Rectangle r?

Do you see the effect of aliasing here?

Suppose we replaced

p := r corner.

with

p := r corner copy.

What would happen then?

Protocol for all objects

Certain messages can be sent to any object. They are defined in the class

Equivalence vs. equality:

• Two objects are equivalent if they are the same object.

== anObject Answer whether the receiver and the argument are the same object.

• Two objects are equal if they have the same value (in some sense).

= anObject Answer whether the receiver and the argument have
the same value.

The decision as to whether the values of two objects are the same is made on a class-by-class basis.

For arrays, equality means—

- are the arrays the same size, \textit{and}
- are corresponding elements equal?

For numbers, equality means both have the same value.

For taxpayers, identity means both have the same social-security number.

Also,

\begin{tabular}{l}
\texttt{~~ not equivalent} \\
\texttt{~\!= not equal}
\end{tabular}

For testing whether a value is \texttt{nil}—

\begin{tabular}{l}
\texttt{isNil} \\
\texttt{notNil}
\end{tabular}

There are also messages for testing the “functionality” of an object:

\begin{tabular}{l}
\texttt{class} \quad \texttt{class of the rcvr.} \\
\texttt{isKindOf: aClass} \quad \texttt{is object of this} \\
\texttt{\quad class or one of its subclasses?} \\
\texttt{isMemberOf: aClass} \quad \texttt{is object of this} \\
\texttt{\quad class?}
\end{tabular}

What is another way of testing

\begin{tabular}{l}
x \texttt{isMemberOf: aClass?} \\
\texttt{x class == aClass.}
\end{tabular}

Let’s try some examples.

\begin{tabular}{l}
aString \texttt{isKindOf: Collection} \\
aSmallInteger \texttt{isKindOf: Number}
\end{tabular}
For cascading —

\[\text{yourself}\]

This message simply returns the receiver. It is useful for creating and initializing complex objects. Suppose we have defined a class \texttt{Polygon} and we want to store its five vertices:

\[
\begin{align*}
\text{myPolygon} & := \text{Polygon new: 5}. \\
\text{myPolygon at: 1 put: myPoint1.} \\
\text{myPolygon at: 2 put: myPoint2.} \\
\text{myPolygon at: 3 put: myPoint3.} \\
\text{myPolygon at: 4 put: myPoint4.} \\
\text{myPolygon at: 5 put: myPoint5.}
\end{align*}
\]

Using cascading, we can shorten this code:

\[
\begin{align*}
\text{myPolygon} & := \text{Polygon new: 5}. \\
\text{myPolygon at: 1 put: myPoint1;} \\
& \quad \text{at: 2 put: myPoint2;} \\
& \quad \text{at: 3 put: myPoint3;} \\
& \quad \text{at: 4 put: myPoint4;} \\
& \quad \text{at: 5 put: myPoint5.}
\end{align*}
\]

A further simplification merges the two statements into one:

\[
\begin{align*}
\text{myPolygon} & := (\text{Polygon new: 5}); \\
& \quad \text{at: 1 put: myPoint1;} \\
& \quad \text{at: 2 put: myPoint2;} \\
& \quad \text{at: 3 put: myPoint3;} \\
& \quad \text{at: 4 put: myPoint4;} \\
& \quad \text{at: 5 put: myPoint5;} \\
& \quad \text{yourself.}
\end{align*}
\]

This is necessary because \texttt{at:put:} returns the value that was inserted.

So without the \texttt{yourself}, the assignment would not be carried out correctly.

\textit{Printing objects}

Methods for “printing” objects return \texttt{Strings}.
Printing returns a representation which is designed to be read. It can either—

- return a string (**printString**), or
- append a string to the argument (**printOn:**).

By default, **printString** returns the object’s class name.

**Example:** a Set of three elements $a, b, and c$ prints as **Set**($a$ $b$ $c$)

**printString** is implemented using **printOn:**. Specifically, it prints on a **WriteStream**. We will discuss streams in the next lecture.

**printString** should not be overridden. Override **printOn:** instead.

For example, the **printOn:** message for class **Character** appends the character $\$$ and then the character that is being printed:

```smalltalk
printOn: aStream
aStream nextPut: $\$$.
aStream nextPut: self
```

**Summary:** • In the Smalltalk class hierarchy, subclasses inherit from superclasses.

- Messages to **super** are used to avoid name conflicts between messages in a class and messages in its superclass(es).
- Abstract superclasses contain the common attributes of related classes, but cannot be used to create instances.
- Instance methods are used to manipulate instances of classes; class methods are used mostly to create instances.
- Each class is the sole member of its metaclass. Metaclass inheritance parallels class inheritance.
• Certain messages are defined on all objects. Among these are messages that test equality and test basic attributes of an object (such as its class).